# Vector and Parallel Algorithms for Integer Factorisation

Richard P. Brent
Computer Sciences Laboratory
Australian National University
Canberra, ACT 2601

**Abstract**

The problem of finding the prime factors of large composite numbers is of practical importance since the advent of public key cryptosystems whose security depends on the presumed difficulty of this problem. In recent years the best known integer factorisation algorithms have improved greatly. It is now routine to factor 60-decimal digit numbers, and possible to factor numbers of more than 110 decimal digits.

We describe several integer factorisation algorithms, and consider their suitability for implementation on vector processors and parallel machines.

## 1. Introduction

Any positive integer $N$ has a unique *prime power decomposition*

$$N = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k} \tag{1.1}$$

($p_1 < p_2 < \cdots < p_k$ primes, $\alpha_j > 0$). To compute the prime power decomposition we need –

1. An algorithm to test if an integer $N$ is prime.
2. An algorithm to find a nontrivial factor $f$ of a composite integer $N$.

Given these components there is a simple recursive algorithm to compute the prime power decomposition (1.1).

*Primality testing*

There are deterministic primality testing algorithms whose worst-case running time on a sequential computer is $O((\log N)^{c \log\log\log N})$, where $c$ is a moderate constant. These algorithms are practical for numbers $N$ of several hundred decimal digits.[12] If we are willing to accept a very small probability of error, then faster (polynomial-time) probabilistic algorithms are available.[6,14] Thus, in this paper we shall assume that primality testing is easy and concentrate on the more difficult problem of factorising composite integers.

*Public key cryptography*

Large primes have at least one practical application – they can be used to construct *public key* cryptosystems (also known as *asymmetric* cryptosystems and *open encryption key* cryptosystems).[29,30] The security of such systems depends on the (assumed) difficulty of factoring the product of two large primes. This is a practical motivation for the current interest in integer factorisation algorithms.

---

Appeared in *Proc. Third Australian Supercomputer Conference* (Melbourne, December 1990).

Copyright © 1990, 3ASC Organising Committee.

E-mail address: rpb@cslab.anu.edu.au

rpb122 typeset using TEX

*Parallel algorithms*

We would hope that an algorithm which required time $T_1$ on a computer with one processor could be implemented to run in time $T_P \sim T_1/P$ on a computer with $P$ independent processors. This is not always the case, since it may be impossible to use all $P$ processors effectively. However, it is true for many integer factorisation algorithms, provided that $P$ is not too large.

The *speedup* of a parallel algorithm is $S = T_1/T_P$ and the *efficiency* is $E = S/P$. We aim for a linear speedup, i.e. $S = \Omega(P)$. If the speedup is linear in the number of processors, then each processor is being used with efficiency bounded below by a positive constant.

## 2. Multiple-Precision Arithmetic

Before describing any integer factorisation algorithms, we comment on the implementation of multiple-precision integer arithmetic on vector processors and parallel machines. Multiple-precision arithmetic is necessary because the number $N$ which we want to factorise or test for primality may be much larger than can be represented in a single computer word.

*Carry propagation and redundant number representations*

To represent a large positive integer $N$, it is customary to choose a convenient *base* or *radix* $\beta$ and express $N$ as

$$N = \sum_{0}^{t-1} d_j \beta^j, \tag{2.1}$$

where $d_0, \ldots, d_{t-1}$ are "base $\beta$ digits" in the range $0 \le d_j < \beta$. We choose $\beta$ large, but small enough that $\beta - 1$ is representable in a single word.[1,14] Consider multiple-precision addition (subtraction and multiplication may be handled in a similar way). On a parallel machine there is a problem with *carry propagation* because a carry can propagate all the way from the least to the most significant digit. Thus an addition takes worst case time $\Omega(t)$, and average time $\Omega(\log t)$, independent of the number of processors.

The carry propagation problem can be reduced if we permit digits $d_j$ outside the normal range. Suppose that we allow $-2 \le d_j \le \beta + 1$, where $\beta > 4$. Then possible carries are in $\{-1, 0, 1, 2\}$ and we need only add corresponding pairs of digits (in parallel), compute the carries and perform one step of carry propagation. (It is only when comparisons of multiple-precision numbers need to be performed that the digits have to be reduced to the normal range by fully propagating carries.) Thus, redundant number representation is useful for speeding up multiple-precision addition and multiplication. On a parallel machine with sufficiently many processors, such a representation allows addition to be performed in constant time.

*High level parallelism*

Rather than try to perform individual multiple-precision operations rapidly, it is often more convenient to implement the multiple-precision operations in bit or word-serial fashion, but perform many independent operations in parallel. For example, a *trial* of the elliptic curve algorithm (Section 7) involves a predetermined sequence of

additions and multiplications on integers of bounded size. Our implementation on a Fujitsu VP 100 performs many trials concurrently in order to take advantage of the machine's vector pipelines.

*Use of real arithmetic*

Most supercomputers were not designed to optimise the performance of exact (multiple-precision) integer arithmetic. On machines with fast floating-point hardware, e.g. pipelined 64-bit floating point units, it may be best to represent base $\beta$ digits in *floating-point* words. The upper bound on $\beta$ is imposed by the multiplication algorithm – we must ensure that $\beta^2$ is exactly representable in a (single or double-precision) floating-point word. For example, on the Fujitsu VP 100, which has a 56-bit fraction and 8-bit exponent, we use $\beta = 2^{26}$.

*Redundant representations* mod $N$

Many integer factorisation algorithms require operations to be performed mod $N$, where $N$ is the number to be factorised. A straightforward implementation would perform a multiple-precision operation and then perform a division by $N$ to find the remainder. However, it is faster to avoid explicit divisions. Usually it is not necessary to represent the result uniquely. For example, consider the computation of $x*y \bmod N$. The result is $r = x*y - q*N$ and it may be sufficient to choose $q$ so that $0 \leq r < 2N$ (a weaker constraint than the usual $0 \leq r < N$). To compute $r$ we multiply $x$ by the digits of $y$, most significant digit first, but modify the standard "shift and add" algorithm to subtract single-precision multiples of $N$ in order to keep the accumulated sum bounded by $2N$. Formally, a partial sum $s$ is updated by $s \leftarrow \beta*s + y_j*x - q_j*N$, where $q_j$ is obtained by a division involving only a few leading digits of $\beta*s + y_j*x$ and $N$.

*Computing inverses* mod $N$

In some factorisation algorithms we need to compute inverses mod $N$. Suppose that $x$ is given, $0 < x < N$, and we want to compute $z$ such that $xz = 1 \bmod N$. The extended Euclidean algorithm[14] applied to $x$ and $N$ gives $u$ and $v$ such that

$$ux + vN = GCD(x, N).$$

If $GCD(x, N) = 1$ then $ux = 1 \bmod N$, so $z = u$. If $GCD(x, N) > 1$ then $GCD(x, N)$ is a nontrivial factor of $N$. This is a rare case where failure (in finding an inverse) implies success (in finding a factor) !

## 3. Integer Factorisation Algorithms

There are many algorithms for finding a nontrivial factor $f$ of a composite integer $N$. The most useful algorithms fall into one of two classes –

A. The run time depends mainly on the size of $N$, and is not strongly dependent on the size of $f$. Examples are –

Lehman's algorithm,[16] which has worst-case run time $O(N^{1/3})$.

The Continued Fraction algorithm[23] and the Multiple Polynomial Quadratic Sieve algorithm,[26] which under plausible assumptions have expected run time $O(\exp(c(\log N \log\log N)^{1/2}))$, where $c$ is a constant (depending on details of the algorithm).

The Number Field Sieve algorithm,[17] which under plausible assumptions has expected run time $O(\exp(c(\log N)^{1/3}(\log\log N)^{2/3}))$, where $c$ is a constant, provided $N$ has a suitable form (see Section 9).

B. The run time depends mainly on the size of $f$, the factor found. (We can assume that $f \le N^{1/2}$.) Examples are –

The trial division algorithm, which has run time $O(f \cdot (\log N)^2)$.

Pollard's "rho" algorithm,[25] which under plausible assumptions has expected run time $O(f^{1/2} \cdot (\log N)^2)$.

Lenstra's Elliptic Curve algorithm,[20] which under plausible assumptions has expected run time $O(\exp(c(\log f \log\log f)^{1/2}) \cdot (\log N)^2)$, where $c$ is a constant.

In these examples, the time bounds are for a sequential machine, and the term $(\log N)^2$ is a generous allowance for the cost of performing arithmetic operations on numbers which are $O(N)$ or $O(N^2)$.

Our survey of integer factorisation algorithms in Sections 4–9 is necessarily cursory. For more information the reader is referred to the literature.[5,6,10,22,26,28]

## 4. Pollard's "rho" algorithm

Pollard's "rho" algorithm[2,25] uses an iteration of the form

$$x_{i+1} = f(x_i) \bmod N, \quad i \geq 0,$$

where $N$ is the number to be factored, $x_0$ is a random starting value, and $f$ is a polynomial with integer coefficients, for example $f(x) = x^2 + a$ ($a \neq 0 \pmod N$).

Let $p$ be the smallest prime factor of $N$, and $j$ the smallest positive index such that $x_{2j} = x_j \pmod p$. Making some plausible assumptions, it is easy to show that the expected value of $j$ is $E(j) = O(p^{1/2})$. The argument is related to the well-known "birthday" paradox – the probability that $x_0, x_1, \ldots, x_k$ are all distinct mod $p$ is approximately $(1-1/p)\cdot(1-2/p)\cdots(1-k/p) \sim \exp(-k^2/(2p))$, and if $x_0, x_1, \ldots, x_k$ are not all distinct mod $p$ then $j \leq k$.

In practice we do not know $p$ in advance, but we can detect $x_j$ by taking greatest common divisors. We simply compute GCD $(x_{2i} - x_i, N)$ for $i = 1, 2, \ldots$ and stop when a GCD greater than 1 is found.

An example of the success of a variation of the Pollard "rho" algorithm is the complete factorisation of the Fermat number $F_8 = 2^{2^8} + 1$ by Brent and Pollard.[8] In fact

$$F_8 = 1238926361552897 \cdot p_{62},$$

where $p_{62}$ is a 62-digit prime.

Parallel implementation of the "rho" algorithm does not give linear speedup. It does not seem possible to use parallelism to speed up the computation of the sequence $(x_i)$ by a significant amount. A plausible use of parallelism is to try several different pseudo-random sequences (generated by different polynomials $f$). If we have $P$ processors and use $P$ different sequences in parallel, the probability that the first $k$ values in each sequence are distinct mod $p$ is approximately $\exp(-k^2P/(2p))$, so the speedup is $O(P^{1/2})$ and the efficiency is only $O(P^{-1/2})$.

## 5. The advantages of a group operation

The Pollard rho algorithm takes $x_{i+1} = f(x_i) \bmod N$ where $f$ is a polynomial. Computing $x_n$ requires $n$ steps. Suppose instead that $x_{i+1} = x_0 * x_i$ where "$*$" is an *associative* operator, *i.e.* $x*(y*z) = (x*y)*z$ . Then we can compute $x_n$ in $O(\log n)$ steps by the *binary powering* method.[14]

Let $m$ be some bound assigned in advance, and let $E$ be the product of all maximal prime powers $q^e$, $q^e \leq m$. Choose some starting value $x_0$, and consider the cyclic group $<x_0>$ consisting of all powers of $x_0$ (under the associative operator "$*$"). If this group has order $g$ whose prime power components are bounded by $m$, then $g|E$ and $x_0^E = I$, where $I$ is the group identity.

We may consider a group defined mod $p$ but work mod $N$, where $p$ is an unknown divisor of $N$. This amounts to using a redundant representation for the group elements. When we compute the identity $I$, its representation mod $N$ may allow us to compute $p$ via a GCD computation (compare Pollard's rho algorithm). We give two examples below: Pollard's $p-1$ algorithm and Lenstra's elliptic curve algorithm.

## 6. Pollard's $p-1$ algorithm

Pollard's "$p-1$" algorithm[24] may be regarded as an attempt to generate the identity in the multiplicative group of $F_p = GF(p)$. Here the group operation "$*$" is just multiplication mod $p$, so (by Fermat's theorem) $g|p-1$ and

$$x_0^E = I \Rightarrow x_0^E = 1 \pmod{p} \Rightarrow p|\text{GCD}\ (x_0^E - 1, N)$$

The "$p-1$" algorithm is very effective in the fortunate case that $p-1$ has no large prime factors. For example, Baillie found the factor

$$p_{25} = 1155685395246619182673033$$

of the Mersenne number $M_{257} = 2^{257} - 1$ (claimed to be prime by Mersenne) using the "$p-1$" algorithm. In this case

$$p_{25} - 1 = 2^3 \cdot 3^2 \cdot 19^2 \cdot 47 \cdot 67 \cdot 257 \cdot 439 \cdot 119173 \cdot 1050151$$

In the worst case, when $(p-1)/2$ is prime, the "$p-1$" algorithm is no better than trial division. Since the group has fixed order $p-1$ there is nothing to be done except try a different algorithm.

Parallel implementation of the "$p-1$" algorithm is difficult, because the inner loop seems inherently serial. At best, parallelism can speed up the multiple precision operations by a small factor (depending on $\log N$ but not on $p$). In the next section we show that it is possible to overcome the main handicaps of the "$p-1$" algorithm, and obtain an algorithm which is easy to implement in parallel and does not depend on a lucky factorisation of $p-1$.

## 7. Lenstra's elliptic curve algorithm

If we can choose a "random" group $G$ with order $g$ close to $p$, we may be able to perform a computation similar to that involved in Pollard's "$p-1$" algorithm, working in $G$ rather than in $F_p$. If all prime factors of $g$ are less than the bound $m$ then we find a factor of $N$. Otherwise, repeat with a different $G$ (and hence, usually, a different $g$) until a factor is found. This is the motivation for H. W. Lenstra's *elliptic curve algorithm* (usually denoted "ECM").

A curve of the form

$$y^2 = x^3 + ax + b \tag{7.1}$$

over some field $F$ is known as an *elliptic curve*. A more general cubic in $x$ and $y$ can be reduced to the form (7.1), which is known as the Weierstrass normal form, by rational transformations.

There is a well-known way of defining an Abelian group $(G, *)$ on an elliptic curve over a field. Formally, if $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ are points on the curve, then the point $P_3 = (x_3, y_3) = P_1 * P_2$ is defined by –

$$(x_3, y_3) = (\lambda^2 - x_1 - x_2,\ \lambda(x_1 - x_3) - y_1), \tag{7.2}$$

where

$$\lambda = \begin{cases} (3x_1^2 + a)/(2y_1) & \text{if } P_1 = P_2 \\ (y_1 - y_2)/(x_1 - x_2) & \text{otherwise.} \end{cases}$$

The identity element $I$ in $G$ is the "point at infinity", $(\infty, \infty)$.

The geometric interpretation of $P_1 * P_2$ is straightforward: the straight line $P_1P_2$ intersects the elliptic curve at a third point $P_3' = (x_3, -y_3)$, and $P_3$ is the reflection of $P_3'$ in the $x$-axis. We refer the reader to a suitable text[13,15] for an introduction to the theory of elliptic curves.

In Lenstra's algorithm[20] the field $F$ is the finite field $F_p$ of $p$ elements, where $p$ is a prime factor of $N$. The multiplicative group of $F_p$, used in Pollard's "$p-1$" algorithm, is replaced by the group $G$ defined by (7.1) and (7.2). Since $p$ is not known in advance, computation is performed in the *ring* $Z/NZ$ of integers modulo $N$ rather than in $F_p$. We can regard this as using a redundant group representation.

A *trial* is the computation involving one random group $G$. The steps involved are –

1. Choose $x_0, y_0$ and $a$ randomly in $[0, N)$. This defines $b = y_0^2 - (x_0^3 + ax_0) \bmod N$. Set $P \leftarrow P_0 = (x_0, y_0)$.
2. For prime $q \le m$ set $P \leftarrow P^{q^e}$ in the group $G$ defined by $a$ and $b$, where $e$ is an exponent chosen as in Section 5. If $P = I$ then the trial succeeds as a factor of $N$ will have been found during an attempt to compute an inverse mod $N$. Otherwise the trial fails.

The work involved in a trial is $O(m)$ group operations. There is a tradeoff involved in the choice of $m$, as a trial with large $m$ is expensive, but a trial with small $m$ is unlikely to succeed.

Given $x \in F_p$, there are at most two values of $y \in F_p$ satisfying (7.1). Thus, allowing for the identity element, we have $g = |G| \le 2p + 1$. A much stronger result, the *Riemann hypothesis for finite fields*, is known –

$$|g - p - 1| < 2p^{1/2}. \tag{7.3}$$

Making the (incorrect, but close enough) assumption that $g$ behaves like a random integer distributed uniformly in $(p - 2p^{1/2}, p + 2p^{1/2})$, we may show that the optimal choice of $m$ is $m = p^{1/\alpha}$, where

$$\alpha \sim (2 \ln p / \ln \ln p)^{1/2} \tag{7.4}$$

It follows that the expected run time is

$$T = p^{2/\alpha + o(1/\alpha)}. \tag{7.5}$$

For details, see Lenstra[20] and Brent.[3] The exponent $2/\alpha$ in (7.5) should be compared with 1 (for trial division) or 1/2 (for Pollard's "rho" method). Because of the overheads involved with ECM, a simpler algorithm such as Pollard's "rho" is preferable for finding factors of size up to about $10^{10}$, but for larger factors the asymptotic advantage of ECM becomes apparent. The following two examples illustrate the power of ECM.

1. A *perfect number* is a number $N$ equal to the sum of its divisors (including 1 but not $N$ itself), e.g. $28 = 1 + 2 + 4 + 7 + 14$. It is not known if any odd perfect numbers exist, but it has been shown[7] that any odd perfect number must be larger than $10^{300}$. The proof required many factorisations of numbers of the form $p^n - 1$, where $p$ and $n$ are prime. Most of these factorisations were found by ECM, e.g. the factorisation

$$4089568263561830388113662969166474269 \cdot p_{65}$$

   of the 101-digit number $(467^{41} - 1)/(466 \cdot 1022869)$.

2. We recently completed the factorisation of the 617-decimal digit Fermat number $F_{11} = 2^{2^{11}} + 1$. In fact

$$F_{11} = 319489 \cdot 974849 \cdot 167988556341760475137 \cdot 3560841906445833920513 \cdot p_{564}$$

   where the 21-digit and 22-digit prime factors were found using ECM, and $p_{564}$ is a 564-decimal digit prime. The factorisation required about 360 million multiplications mod $N$, which took less than 2 hours on a Fujitsu VP 100 vector processor.[4]

*A second phase*

Both the Pollard "$p - 1$" and Lenstra elliptic curve algorithms can be speeded up by the addition of a second phase. The idea of the second phase is to find a factor in the case that the first phase terminates with a group element $P \neq I$, such that $|\langle P \rangle|$ is reasonably small (say $O(m^2)$). (Here $\langle P \rangle$ is the cyclic group generated by $P$.) There are several possible implementations of the second phase. One of the simplest uses a pseudorandom walk in $\langle P \rangle$. By the birthday paradox argument, there is a good chance that two points in the random walk will coincide after $O(|\langle P \rangle|)^{1/2}$ steps, and when this occurs a nontrivial factor of $N$ can usually be found. Details may be found in Brent[3] and Montgomery.[22]

The use of a second phase provides a significant speedup in practice, but does not change the asymptotic time bound (7.5). Similar comments apply to other implementation details, such as ways of avoiding most divisions and speeding up group

operations, ways of choosing good initial points, and ways of using preconditioned polynomial evaluation.[3,21,22]

*Parallel implementation of ECM*

So long as the expected number of trials is much larger than the number $P$ of processors available, linear speedup is possible by performing $P$ trials in parallel. In fact, if $T_1$ is the expected run time on one processor, then the expected run time on an MIMD parallel machine with $P$ processors is

$$T_P = T_1/P + O(T_1^{1/2+\epsilon}) \tag{7.6}$$

The bound (7.6) applies on single-instruction multiple-data (SIMD) machines if we use the Montgomery-Chudnovsky form $by^2 = x^3+ax^2+x$ instead of the Weierstrass normal form (7.1) in order to avoid divisions.[22]

In practice, it may be difficult to perform $P$ trials in parallel because of storage limitations. The second phase requires much more storage than the first phase. Fortunately, there are several possibilities for making use of parallelism during the second phase of each trial. Our implementation performs the first phase of $P$ trials in parallel, but the second phase of each trial sequentially, using $P$ processors to speed up the evaluation of the high-degree polynomials which constitute most of the work during the second phase.

## 8. Quadratic sieve algorithms

Quadratic sieve algorithms belong to a wide class of algorithms which try to find two integers $x$ and $y$ such that $x \neq \pm y \pmod N$ but

$$x^2 = y^2 \pmod N \tag{8.1}$$

Once such $x$ and $y$ are found, then GCD $(x-y, N)$ is a nontrivial factor of $N$.

One way to find $x$ and $y$ satisfying (8.1) is to find a set of relations of the form

$$u_i^2 = v_i^2 w_i \pmod N, \tag{8.2}$$

where the $w_i$ have all their prime factors in a moderately small set of primes (called the *factor base*). Each relation (8.2) gives a row in a matrix $M$ whose columns correspond to the primes in the factor base. Once enough rows have been generated, we can use sparse Gaussian elimination in $F_2$ (Weidemann[33]) to find a linear dependency (mod 2) between a set of rows of $M$. Multiplying the corresponding relations now gives a relation of the form (8.1).

In quadratic sieve algorithms the numbers $w_i$ are the values of one (or more) polynomials with integer coefficients. This makes it easy to factorise the $w_i$ by *sieving*. For details of the process, we refer to several recent papers.[11,19,26,27,31] The inner loop of the sieving process looks like

**while** $j < bound$ **do**
**begin**
$s[j] \leftarrow s[j] + c$;
$j \leftarrow j + q$;
**end**

Here *bound* depends on the size of the (single-precision real) sieve array $s$, $q$ is a small prime or prime power, and $c$ is a single-precision real constant depending on $q$ ($c = \Lambda(q) = \log p$ if $q = p^e$, $p$ prime). The loop can be implemented efficiently on a pipelined vector processor. It is possible to use scaling to avoid floating point additions, which is desirable on a small processor without floating-point hardware.

The best quadratic sieve algorithms such as the *multiple polynomial quadratic sieve algorithm* MPQS[26] can, under plausible assumptions, factor a number $N$ in time $O(\exp(c(\log N \log\log N)^{1/2}))$, where $c \sim 1$. The constants involved are such that MPQS is usually faster than ECM if $N$ is the product of two primes which both exceed $N^{1/3}$.

At first sight it is surprising that algorithms as different as MPQS and ECM have similar expected time bounds. MPQS requires $O(B)$ factorisations of numbers $w_i$ of size $O(N^{1/2+\epsilon})$ over the factor base of size $B$, and the work per trial is small (because of the efficiency of the sieving process). On the other hand, ECM requires only one number (the order of the group $G$) to factor completely over primes not exceeding $m$, but the work per trial is $O(m)$. Use of "partial relations", i.e. incompletely factored $w_i$, in MPQS is analogous to the second phase of ECM and gives a similar performance improvement.

*Parallel implementation of MPQS*

Like ECM, MPQS is ideally suited to parallel implementation. Different processors may use different polynomials, or sieve over different intervals with the same polynomial. Thus, there is a linear speedup so long as the number of processors is not much larger than the size of the factor base. The process requires very little communication between processors. Each processor can generate relations and forward them to some central collection point. This has been demonstrated most clearly by A. K. Lenstra and M. S. Manasse[19] who distribute their program and collect relations via electronic mail. The processors are scattered around the world – anyone with access to electronic mail and a C compiler can volunteer to contribute. The final stage – Gaussian elimination to combine the relations – is not so easily distributed. However, in practice it is only a small fraction of the computation.

MPQS has been used to obtain many spectacular factorisations.[9,27,31] Lenstra and Manasse[19] (with many assistants scattered around the world) have factorised several numbers larger than $10^{100}$. For example, a recent factorisation was the 111-decimal digit number

$$2^{484} + 1 = 3860897986942821068655933036263824535533549879744 1 \cdot p_{61}$$

Such factorisations require many years of CPU time, but a real time of only a month or so because of the number of different processors which are working in parallel.

## 9. The number field sieve (NFS)

Our numerical examples have all involved numbers of the form

$$a^e \pm b, \tag{9.1}$$

for small $a$ and $b$, although the ECM and MPQS factorisation algorithms do not take advantage of this special form.

The *number field sieve* (NFS) is a new algorithm which does take advantage of the special form (9.1). In concept it is similar to the quadratic sieve algorithm, but it works over an algebraic number field defined by $a$, $e$ and $b$. We refer the interested reader to Lenstra *et al* [17] for details, and merely give an example to show the power of the algorithm. For an introduction to the relevant concepts of algebraic number theory, see Stewart and Tall.[32]

Consider the 155-decimal digit number

$$F_9 = N = 2^{2^9} + 1$$

as a candidate for factoring by NFS. Note that $8N = m^5 + 8$, where $m = 2^{103}$. We may work in the number field $Q(\alpha)$, where $\alpha$ satisfies

$$\alpha^5 + 8 = 0,$$

and in the ring of integers of $Q(\alpha)$. Because

$$m^5 + 8 = 0 \pmod{N},$$

the mapping $\phi : \alpha \mapsto m \bmod N$ is a ring homomorphism from $Z[\alpha]$ to $Z/NZ$.

The idea is to search for pairs of small coprime integers $u$ and $v$ such that both the algebraic integer $u + \alpha v$ and the (rational) integer $u + mv$ can be factorised. (The factor base now includes prime ideals and units as well as rational primes.) Because

$$\phi(u + \alpha v) = (u + mv) \pmod{N},$$

each such pair gives a relation analogous to (8.2).

The prime ideal factorisation of $u + \alpha v$ can be obtained from the factorisation of the *norm* $u^5 - 8v^5$ of $u + \alpha v$. Thus, we have to factor simultaneously two integers $u + mv$ and $|u^5 - 8v^5|$. Note that, for moderate $u$ and $v$, both these integers are much smaller than $N$, in fact they are $O(N^{1/d})$, where $d = 5$ is the degree of the algebraic number field. (The optimal choice of $d$ is discussed in Lenstra *et al*.[17])

Using these and related ideas, Lenstra *et al* [18] recently factorised $F_9$, obtaining

$$F_9 = 2424833 \cdot 7455602825647884208337395736200454918783366342657 \cdot p_{99},$$

where $p_{99}$ is an 99-digit prime, and the 7-digit factor was already known. The collection of relations took less than two months on a network of several hundred workstations. A sparse system of about 200,000 relations was reduced to a dense matrix with about 72,000 rows. Using Gaussian elimination, dependencies (mod 2) between the rows were found in three hours on a Connection Machine. These dependencies implied equations of the form $x^2 = y^2 \bmod F_9$. The second such equation was nontrivial and gave the desired factorisation of $F_9$.

## 10. Conclusion

We have sketched some algorithms for integer factorisation. The algorithms draw on results in elementary number theory, algebraic number theory and probability theory. As well as their inherent interest and applicability to other areas of mathematics, advances in computing technology have given them practical applications in the area of secure communications.

Despite much progress in the development of efficient algorithms, our knowledge of the complexity of factorisation is inadequate. We would like to find a polynomial time factorisation algorithm or else prove that one does not exist. Until a polynomial time algorithm is found, large factorisations will continue to challenge the capabilities of supercomputers.


## Acknowledgement

Some of the numerical results quoted above were obtained with the assistance of the Australian National University Supercomputer Facility.


## References


1. R. P. Brent, "A Fortran multiple-precision arithmetic package", *ACM Transactions on Mathematical Software* 4 (1978), 57-70.
2. R. P. Brent, "An improved Monte Carlo factorisation algorithm", *BIT* 20 (1980), 176-184.
3. R. P. Brent, "Some integer factorisation algorithms using elliptic curves", *Australian Computer Science Communications* 8 (1986), 149-163.
4. R. P. Brent, "Factorisation of the eleventh Fermat number (preliminary report)", *AMS Abstracts* 10 (1989), 89T-11-73.
5. R. P. Brent, "Parallel algorithms for integer factorisation", in *Number Theory and Cryptography* (edited by J. H. Loxton), London Mathematical Society Lecture Note Series 154, Cambridge University Press, 1990, 26-37.
6. R. P. Brent, "Primality testing and integer factorisation", *The Role of Mathematics in Science*, Proceedings of a Symposium held at the Australian Academy of Science, Canberra, April 1990, to appear.
7. R. P. Brent, G. L. Cohen and H. J. J. te Riele, "Improved techniques for lower bounds for odd perfect numbers", to appear in *Mathematics of Computation*.
8. R. P. Brent and J. M. Pollard, "Factorisation of the eighth Fermat number", *Mathematics of Computation* 36 (1981), 627-630.
9. J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman and S. S. Wagstaff, Jr., *Factorisations of* $b^n \pm 1, b = 2, 3, 5, 6, 7, 10, 11, 12$ *up to high powers*, American Mathematical Society, Providence, Rhode Island, second edition, 1985.
10. D. A. Buell, "Factoring: algorithms, computations, and computers", *J. Supercomputing* 1 (1987), 191-216.
11. T. R. Caron and R. D. Silverman, "Parallel implementation of the quadratic sieve", *J. Supercomputing* 1 (1988), 273-290.
12. H. Cohen and H. W. Lenstra, Jr., "Primality testing and Jacobi sums", *Mathematics of Computation* 42 (1984), 297-330.
13. K. F. Ireland and M. Rosen, *A Classical Introduction to Modern Number Theory*, Springer-Verlag, 1982, Ch. 18.

14. D. E. Knuth, *The Art of Computer Programming*, Vol. 2, Addison Wesley, 2nd edition, 1982.
15. S. Lang, *Elliptic Curves – Diophantine Analysis*, Springer-Verlag, 1978.
16. R. S. Lehman, "Factoring large integers", *Mathematics of Computation* 28 (1974), 637-646.
17. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse and J. M. Pollard, *The number field sieve*, Proc. 22nd Annual ACM Conference on Theory of Computing, Baltimore, Maryland, May 1990, 564-572.
18. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse and J. M. Pollard, *Complete factorization of the ninth Fermat number*, e-mail announcement, 15 June 1990.
19. A. K. Lenstra and M. S. Manasse, "Factoring by electronic mail", to appear in *Proceedings Eurocrypt '89.*
20. H. W. Lenstra, Jr., "Factoring integers with elliptic curves", *Ann. of Math.* (2) 126 (1987), 649-673.
21. P. L. Montgomery, "Modular multiplication without trial division", *Mathematics of Computation* 44 (1985), 519-521.
22. P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorisation", *Mathematics of Computation* 48 (1987), 243-264.
23. M. A. Morrison and J. Brillhart, "A method of factorisation and the factorisation of $F_7$", *Mathematics of Computation* 29 (1975), 183-205.
24. J. M. Pollard, "Theorems in factorisation and primality testing", *Proc. Cambridge Philos. Soc.* 76 (1974), 521-528.
25. J. M. Pollard, "A Monte Carlo method for factorisation", *BIT* 15 (1975), 331-334.
26. C. Pomerance, J. W. Smith and R. Tuler, "A pipeline architecture for factoring large integers with the quadratic sieve algorithm", *SIAM J. on Computing* 17 (1988), 387-403.
27. H. J. J. te Riele, W. Lioen and D. Winter, "Factoring with the quadratic sieve on large vector computers", *Belgian J. Comp. Appl. Math.* 27(1989), 267-278.
28. H. Riesel, *Prime Numbers and Computer Methods for Factorisation*, Birkhäuser, Boston, 1985.
29. R. L. Rivest, A. Shamir and L. Adelman, "A method for obtaining digital dignatures and public-key cryptosystems", *Communications of the ACM* 21 (1978), 120-126.
30. J. Seberry and J. Pieprzyk, *Cryptography: An Introduction to Computer Security*, Prentice Hall, Sydney, 1989.
31. R. D. Silverman, "The multiple polynomial quadratic sieve", *Mathematics of Computation* 48 (1987), 329-339.
32. I. N. Stewart and D. O. Tall, *Algebraic Number Theory*, second edition, Chapman and Hall, 1987.
33. D. Wiedemann, "Solving sparse linear equations over finite fields", *IEEE Trans. Inform. Theory* 32 (1986), 54-62.